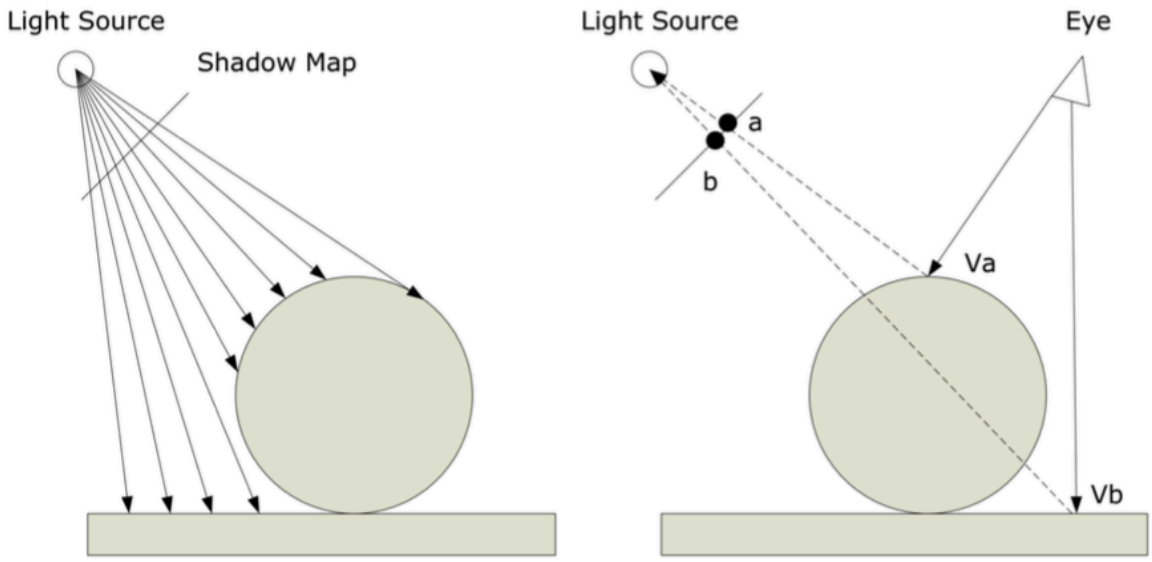
Homework 7 - Shadowing Mapping

1. 阴影渲染



阴影渲染两大基本步骤:

- 以光源视角渲染场景，得到深度图(DepthMap)，并存储为texture;
- 以camera视角渲染场景，使用Shadowing Mapping算法(比较当前深度值与在DepthMap Texture的深度 值)，决定某个点是否在阴影下。

2. 阴影优化
   - 阴影锯齿消除
   - Common Techniques to Improve Shadow Depth Maps - Microsoft

Homework

Basic:

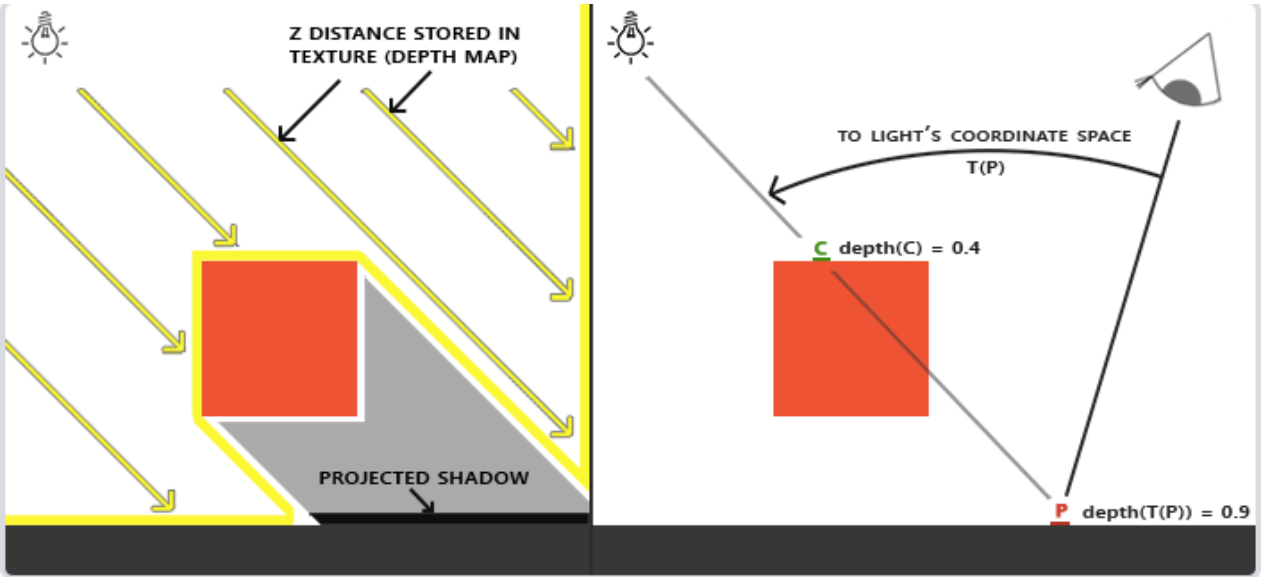
1. 实现方向光源的Shadowing Mapping:
   - 要求场景中至少有一个object和一块平面(用于显示shadow)
   - 光源的投影方式任选其一即可
   - 在报告里结合代码，解释Shadowing Mapping算法
2. 修改GUI

Bonus:

1. 实现光源在正交/透视两种投影下的Shadowing Mapping
2. 优化Shadowing Mapping (可结合References链接，或其他方法。优化方式越多越好，在报告里说明，有加分)

Answer

阴影映射



左侧的图片展示了一个定向光源（所有光线都是平行的）在立方体下的表面投射的阴影。通过储存到深度贴图中的深度值，我们就能找到最近点，用以决定片元是否在阴影中。我们使用一个来自光源的视图和投影矩阵来渲染场景就能创建一个深度贴图。这个投影和视图矩阵结合在一起成为一个TT变换，它可以将任何三维位置转变到光源的可见坐标空间。

定向光并没有位置，因为它被规定为无穷远。然而，为了实现阴影贴图，我们得从一个光的透视图渲染场景，这样就得在光的方向的某一点上渲染场景。

在右边的图中我们显示出同样的平行光和观察者。我们渲染一个点$\bar{P}$处的片元，需要决定它是否在阴影中。我们先得使用TT把$\bar{P}$变换到光源的坐标空间里。既然点$\bar{P}$是从光的透视图中看到的，它的z坐标就对应于它的深度，例子中这个值是0.9。使用点$\bar{P}$在光源的坐标空间的坐标，我们可以索引深度贴图，来获得从光的视角中最近的可见深度，结果是点$\bar{C}$，最近的深度是0.4。因为索引深度贴图的结果是一个小于点$\bar{P}$的深度，我们可以断定$\bar{P}$被挡住了，它在阴影中了。

step1：深度贴图

为渲染的深度贴图创建一个帧缓冲对象：

```
GLuint depthMapFBO;
glGenFramebuffers(1, &depthMapFBO);
```

创建一个2D纹理，提供给帧缓冲的深度缓冲使用：

```
const GLuint SHADOW_WIDTH = 1024, SHADOW_HEIGHT = 1024;

GLuint depthMap;
glGenTextures(1, &depthMap);
glBindTexture(GL_TEXTURE_2D, depthMap);
glTexImage2D(GL_TEXTURE_2D, 0, GL_DEPTH_COMPONENT,
             SHADOW_WIDTH, SHADOW_HEIGHT, 0, GL_DEPTH_COMPONENT, GL_FLOAT,
NULL);
glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_MIN_FILTER, GL_NEAREST);
glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_MAG_FILTER, GL_NEAREST);
glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_WRAP_S, GL_REPEAT);
glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_WRAP_T, GL_REPEAT);
```

只需关心深度值，所以把纹理格式指定为GL_DEPTH_COMPONENT。

把生成的深度纹理作为帧缓冲的深度缓冲：

```
glBindFramebuffer(GL_FRAMEBUFFER, depthMapFBO);
glFramebufferTexture2D(GL_FRAMEBUFFER, GL_DEPTH_ATTACHMENT, GL_TEXTURE_2D,
depthMap, 0);
glDrawBuffer(GL_NONE);
glReadBuffer(GL_NONE);
glBindFramebuffer(GL_FRAMEBUFFER, 0);
```

调用glDrawBuffer和glReadBuffer把读和绘制缓冲设置为GL_NONE，显式告诉OpenGL不适用任何颜色数据进行渲染。

合理配置将深度值渲染到纹理的帧缓冲后，(1)生成深度贴图，(2)用深度贴图渲染场景：

```
// 1. 首选渲染深度贴图
glViewport(0, 0, SHADOW_WIDTH, SHADOW_HEIGHT);
glBindFramebuffer(GL_FRAMEBUFFER, depthMapFBO);
glClear(GL_DEPTH_BUFFER_BIT);
ConfigureShaderAndMatrices();
RenderScene();
glBindFramebuffer(GL_FRAMEBUFFER, 0);
// 2. 像往常一样渲染场景，但这次使用深度贴图
glViewport(0, 0, SCR_WIDTH, SCR_HEIGHT);
glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT);
  ...    // 为每个物体设置合适的投影和视图矩阵，以及相关的模型矩阵
glBindTexture(GL_TEXTURE_2D, depthMap);
RenderScene();
```

`glViewport`函数可以改变视口（viewport）的参数以适应阴影贴图的尺寸。

step2：光源空间的变换

实验采用的是所有光线平行的定向光。

```
glm::mat4 lightProjection, lightView;
glm::mat4 lightSpaceMatrix;
GLfloat near_plane = 1.0f, far_plane = 7.5f;
if(mode == 0)
    lightProjection = glm::ortho(-10.0f, 10.0f, -10.0f, 10.0f, near_plane,
far_plane);
else
    lightProjection = glm::perspective(fov, (GLfloat)SHADOW_WIDTH /
(GLfloat)SHADOW_HEIGHT, near_plane, far_plane);
lightView = glm::lookAt(lightPos, glm::vec3(0.0f), glm::vec3(0.0, 1.0, 0.0));
lightSpaceMatrix = lightProjection * lightView;
```

光源可以选择采用正交或者透视投影。正交投影矩阵并不会将场景用透视图进行变形，所有视线/光线都是平行的，这使它对于定向光来说是个很好的投影矩阵。然而透视投影矩阵，会将所有顶点根据透视关系进行变形，结果因此而不同。

将`lightView`和`lightProjection`两矩阵的乘积作为光空间的变换矩阵`lightSpaceMatrix`，它将每个世界空间坐标变换到光源处所见到的那个空间。这个lightSpaceMatrix正是前面我们称为TT的那个变换矩阵。有了lightSpaceMatrix只要给shader提供光空间的投影和视图矩阵，我们就能像往常那样渲染场景了。

着色器

- `shadow_mapping_depth`

  shadow_mapping_depth.vs:

  ```
  #version 330 core
  layout (location = 0) in vec3 aPos;

  uniform mat4 lightSpaceMatrix;
  uniform mat4 model;

  void main()
  {
      gl_Position = lightSpaceMatrix * model * vec4(aPos, 1.0);
  }
  ```

  这个顶点着色器将一个单独模型的一个顶点，使用lightSpaceMatrix变换到光空间中。

  shadow_mapping_depth.fs:

  ```
  #version 330 core

  void main()
  {
      // gl_FragDepth = gl_FragCoord.z;
  }
  ```

没有颜色缓冲，所有片段着色器不需要任何转换。

注释部分可以用来显式设置深度。

- `shadow_mapping`

  shadow_mapping.vs:

```
#version 330 core
layout (location = 0) in vec3 position;
layout (location = 1) in vec3 normal;
layout (location = 2) in vec2 texCoords;

out VS_OUT {
    vec3 FragPos;
    vec3 Normal;
    vec2 TexCoords;
    vec4 FragPosLightSpace;
} vs_out;

uniform mat4 projection;
uniform mat4 view;
uniform mat4 model;
uniform mat4 lightSpaceMatrix;

void main()
{
    gl_Position = projection * view * model * vec4(position, 1.0f);
    vs_out.FragPos = vec3(model * vec4(position, 1.0));
    vs_out.Normal = transpose(inverse(mat3(model))) * normal;
    vs_out.TexCoords = texCoords;
    vs_out.FragPosLightSpace = lightSpaceMatrix * vec4(vs_out.FragPos,
1.0);
}
```

  相同的，把世界空间顶点位置转换为光空间。

  顶点着色器传递一个普通的经变换的世界空间顶点位置vs_out.FragPos和一个光空间的vs_out.FragPosLightSpace给片段着色器。

  shadow_mapping.fs:

```
#version 330 core
out vec4 FragColor;

in VS_OUT {
    vec3 FragPos;
    vec3 Normal;
    vec2 TexCoords;
    vec4 FragPosLightSpace;
} fs_in;
```

```
uniform sampler2D diffuseTexture;
uniform sampler2D shadowMap;

uniform vec3 lightPos;
uniform vec3 viewPos;

uniform bool shadows;

float ShadowCalculation(vec4 fragPosLightSpace)
{
    // 执行透视除法
    vec3 projCoords = fragPosLightSpace.xyz / fragPosLightSpace.w;
    [···]
    return shadow;
}

void main()
{
    vec3 color = texture(diffuseTexture, fs_in.TexCoords).rgb;
    vec3 normal = normalize(fs_in.Normal);
    vec3 lightColor = vec3(0.4);
    // Ambient
    vec3 ambient = 0.2 * color;
    // Diffuse
    vec3 lightDir = normalize(lightPos - fs_in.FragPos);
    float diff = max(dot(lightDir, normal), 0.0);
    vec3 diffuse = diff * lightColor;
    // Specular
    vec3 viewDir = normalize(viewPos - fs_in.FragPos);
    float spec = 0.0;
    vec3 halfwayDir = normalize(lightDir + viewDir);
    spec = pow(max(dot(normal, halfwayDir), 0.0), 64.0);
    vec3 specular = spec * lightColor;
    // Calculate shadow
    float shadow = shadows ? ShadowCalculation(fs_in.FragPosLightSpace) :
0.0;
    shadow = min(shadow, 0.75); // reduce shadow strength a little: allow
some diffuse/specular light in shadowed regions
    vec3 lighting = (ambient + (1.0 - shadow) * (diffuse + specular)) *
color;

    FragColor = vec4(lighting, 1.0f);
}
```

片段着色器使用Blinn-Phong光照模型渲染场景。我们接着计算出一个shadow值，当fragment在阴影中时是1.0，在阴影外是0.0。然后，diffuse和specular颜色会乘以这个阴影元素。由于阴影不会是全黑的（由于散射），我们把ambient分量从乘法中剔除。

声明了一个shadowCalculation函数，用于计算阴影。首先要检查一个片元是否在阴影中，把光空间片元位置转换为裁切空间的标准化设备坐标。当我们在顶点着色器输出一个裁切空间顶点位置到gl_Position时，OpenGL自动进行一个透视除法，将裁切空间坐标的范围-w到w转为-1到1，这要将x、y、z元素除以向量的w元素来实现。

上面的projCoords的xyz分量都是[-1,1]（下面会指出这对于远平面之类的点才成立），而为了和深度贴图的深度相比较，z分量需要变换到[0,1]；为了作为从深度贴图中采样的坐标，xy分量也需要变换到[0,1]。所以整个projCoords向量都需要变换到[0,1]范围。

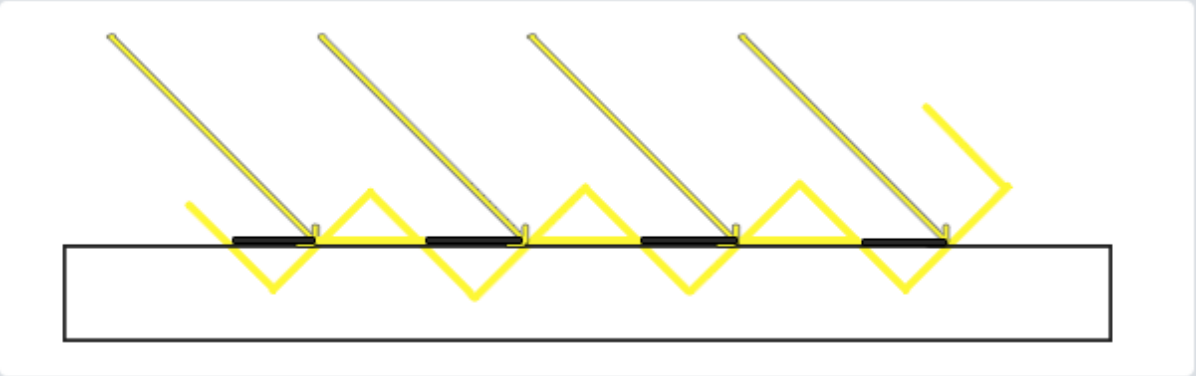
```
projCoords = projCoords * 0.5 + 0.5;
```

然后需要将当前片段深度和缓存中对应位置深度比较，确定该片段是否在阴影中：

```
// 取得最近点的深度(使用[0,1]范围下的fragPosLight当坐标)
float closestDepth = texture(shadowMap, projCoords.xy).r;
// 取得当前片元在光源视角下的深度
float currentDepth = projCoords.z;
// 检查当前片元是否在阴影中
float shadow = currentDepth > closestDepth  ? 1.0 : 0.0;
```

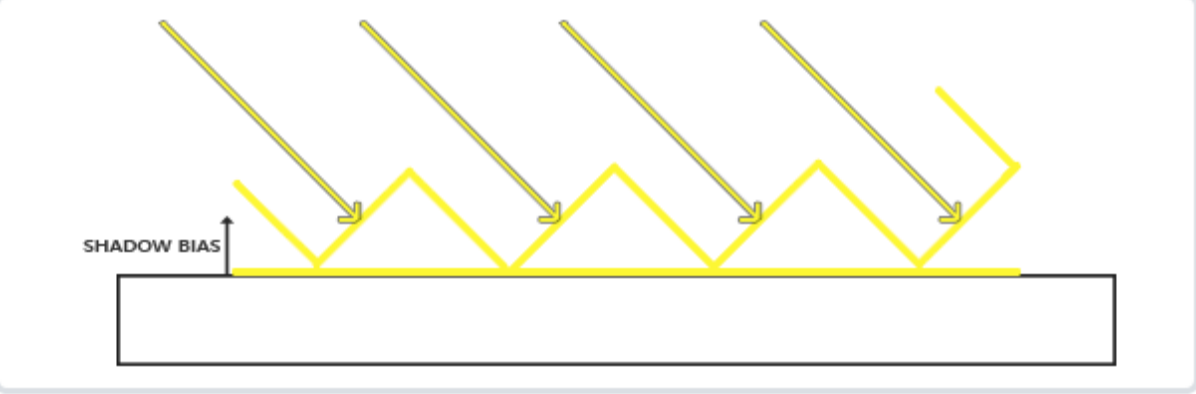
改进

- 阴影失真



因为阴影贴图受限于解析度，在距离光源比较远的情况下，多个片元可能从深度贴图的同一个值中去采样。图片每个斜坡代表深度贴图一个单独的纹理像素。你可以看到，多个片元从同一个深度值进行采样。

我们可以用一个叫做**阴影偏移**（shadow bias）的技巧来解决这个问题，我们简单的对表面的深度（或深度贴图）应用一个偏移量，这样片元就不会被错误地认为在表面之下了。



解决：根据表面朝向光线的角度更改偏移量，使用点乘：

```
float bias = max(0.05 * (1.0 - dot(normal, lightDir)), 0.005);
```

这里有一个偏移量的最大值0.05，和一个最小值0.005，它们是基于表面法线和光照方向的。

- 悬浮

  阴影相对实际物体位置的偏移叫悬浮。

  解决：进行正面剔除，先必须开启GL_CULL_FACE：

```
glCullFace(GL_FRONT);
RenderSceneToDepthMap();
glCullFace(GL_BACK); // 不要忘记设回原先的culling face
```

- 采样过多

  光照有一个区域，超出该区域就成为了阴影；这个区域实际上代表着深度贴图的大小，这个贴图投影到了地板上。发生这种情况的原因是我们之前将深度贴图的环绕方式设置成了GL_REPEAT。

  解决：可以储存一个边框颜色，然后把深度贴图的纹理环绕选项设置为GL_CLAMP_TO_BORDER：

```
glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_WRAP_S, GL_CLAMP_TO_BORDER);
glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_WRAP_T, GL_CLAMP_TO_BORDER);
GLfloat borderColor[] = { 1.0, 1.0, 1.0, 1.0 };
glTexParameterfv(GL_TEXTURE_2D, GL_TEXTURE_BORDER_COLOR, borderColor);
```

  投影向量的z坐标大于1.0时，把shadow的值强制设为0.0：

```
if(projCoords.z > 1.0)
    shadow = 0.0;
```

- PCF

  PCF（percentage-closer filtering），是一种多个不同过滤方式的组合，它产生柔和阴影，使它们出现更少的锯齿块和硬边。核心思想是从深度贴图中多次采样，每一次采样的纹理坐标都稍有不同。每个独立的样本可能在也可能不再阴影中。所有的次生结果接着结合在一起，进行平均化，我们就得到了柔和阴影。

```
float shadow = 0.0;
vec2 texelSize = 1.0 / textureSize(shadowMap, 0);
for(int x = -1; x <= 1; ++x)
{
    for(int y = -1; y <= 1; ++y)
    {
        float pcfDepth = texture(shadowMap, projCoords.xy + vec2(x, y) *
texelSize).r;
        shadow += currentDepth - bias > pcfDepth ? 1.0 : 0.0;
    }
}
shadow /= 9.0;
```

Shadowing Mapping算法原理

判断片段是否处在阴影中，分三步：

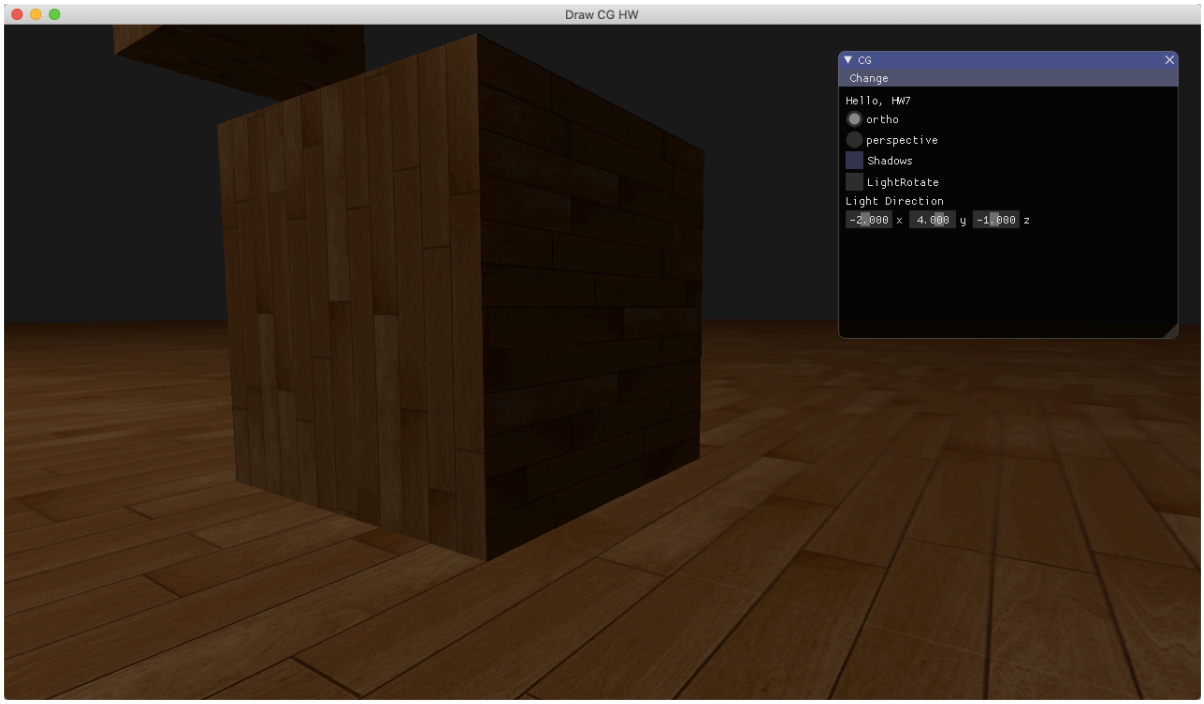
1. 从光源的位置对场景进行渲染，得到一张深度图p1。其中的每一个元素都记录了第一个可见表面的深度值（与光源的距离）。
2. 从真实的视点对场景进行渲染，得到一张深度图p2。
3. 对深度图p2逐片元（如上图中的B）地通过矩阵变换，切换到光源空间下，与深度图p1相应位置（如上图中的A）的深度进行比较。如果大于p1相应位置的深度（距离光源更远），那么说明p2上的此片元（如上图中的B）应该是阴影。否则，被照亮。

IMGUI：

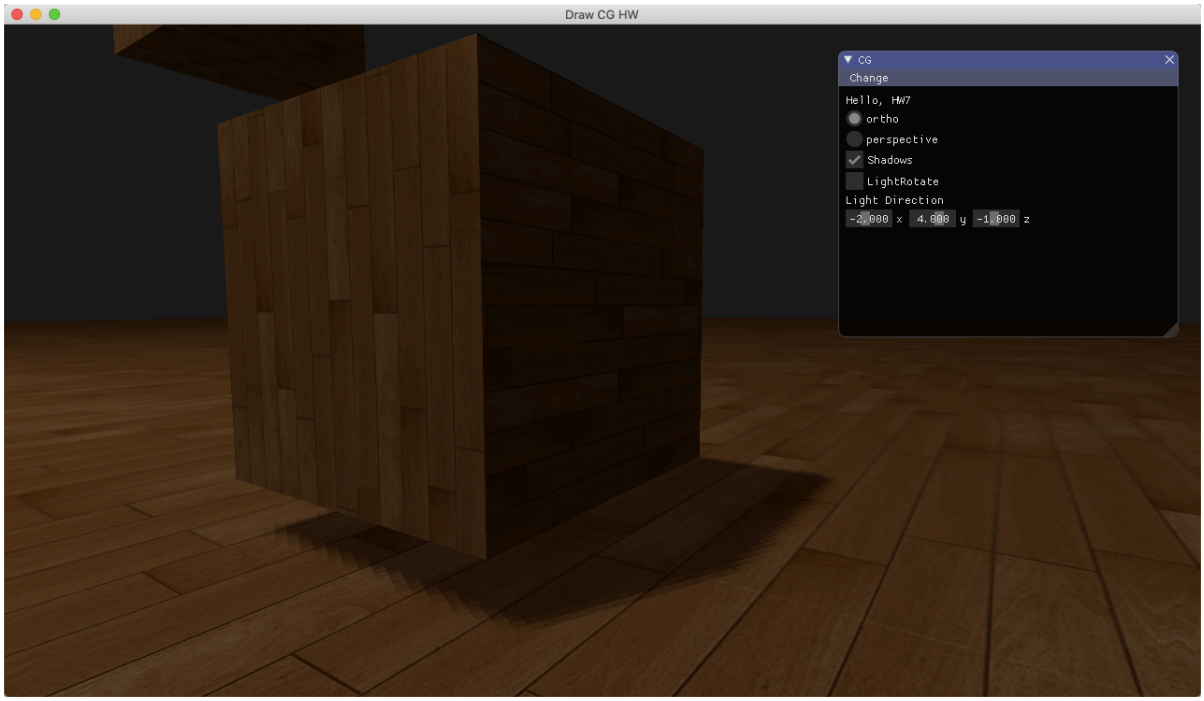
```
ImGui::RadioButton("ortho", &chooseItem, 0);
ImGui::RadioButton("perspective", &chooseItem, 1);

ImGui::Checkbox("Shadows", &shadows);
if(chooseItem ==1){
    ImGui::PushItemWidth(100);
    ImGui::SliderFloat("Presp fov", &fov, -90.0f, 90.0f);
}

ImGui::Text("Light Direction");
ImGui::PushItemWidth(50);
ImGui::SliderFloat("x", &lightPos[0], -10.0f, 10.0f);
ImGui::SameLine();
ImGui::SliderFloat("y", &lightPos[1], -10.0f, 10.0f);
ImGui::SameLine();
ImGui::SliderFloat("z", &lightPos[2], -10.0f, 10.0f);
```

实现效果

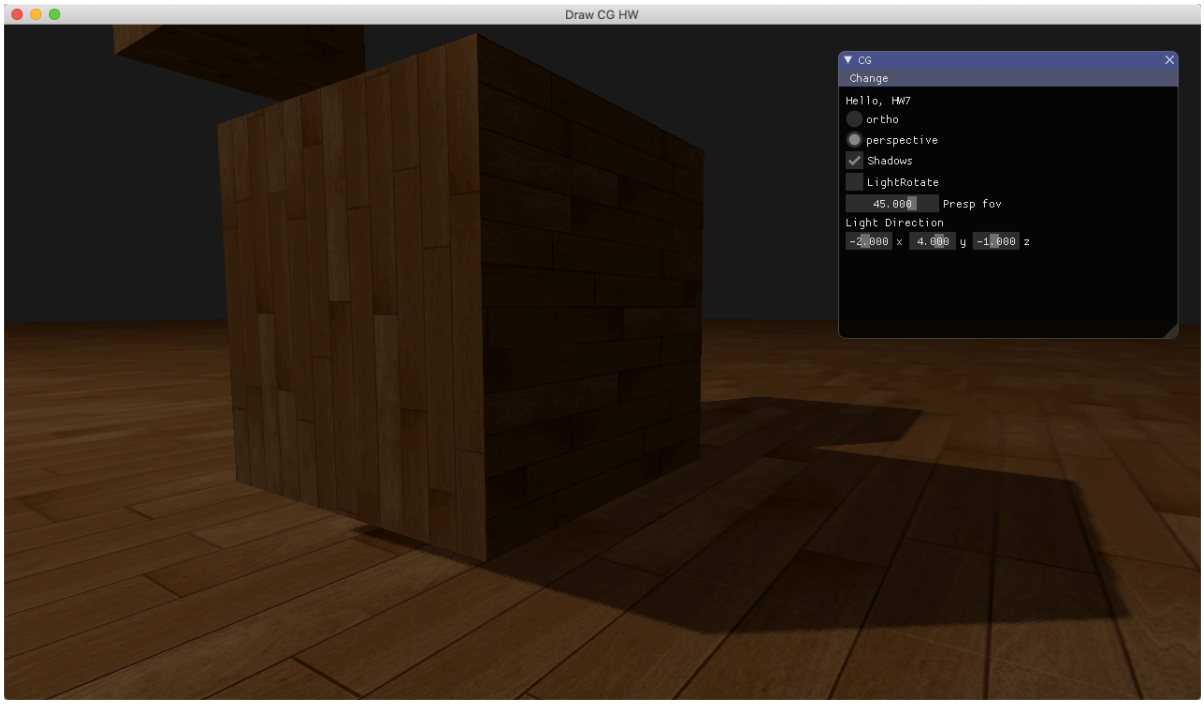
- 无阴影情况


Draw CG HW
CG
Change
Hello, HW7
ortho
perspective
Shadows
LightRotate
Light Direction
-2.000 x 4.000 y -1.000 z

- 正射投影


Draw CG HW
CG
Change
Hello, HW7
ortho
perspective
Shadows
LightRotate
Light Direction
-2.000 x 4.000 y -1.000 z

- 透视投影


Draw CG HW
CG
Change
Hello, HW7
ortho
perspective
Shadows
LightRotate
45.000
Presp fov
Light Direction
-2.000 x
4.000 y
-1.000 z

- 光线方向旋转